Functional Baby Talk: Analysis of Code Fragments from Novice Haskell Programmers

Jeremy Singer and Blair Archibald
School of Computing Science
University of Glasgow
UK
jeremy.singer@glasgow.ac.uk b.archibald.1@research.gla.ac.uk

What kinds of mistakes are made by novice Haskell developers, as they learn about functional programming? Is it possible to analyze these errors in order to improve the pedagogy of Haskell? In 2016, we delivered a massive open online course which featured an interactive code evaluation environment. We captured and analyzed 161K interactions from learners. We report typical novice developer behavior; for instance, the mean time spent on an interactive tutorial is around eight minutes. Although our environment was restricted, we gain some understanding of Haskell novice errors. Parenthesis mismatches, lexical scoping errors and `do` block misunderstandings are common. Finally, we make recommendations about how such beginner code evaluation environments might be enhanced.

1 Introduction

The Haskell programming language [14] has acquired a reputation for being difficult to learn. In his presentation on the origins of Haskell [23] Peyton Jones notes that, according to various programming language popularity metrics, Haskell is much more frequently discussed than it is used for software implementation. The xkcd comic series features a sarcastic strip on Haskell's side-effect free property [21]. Haskell code is free from side-effects 'because no-one will ever run it.'

In 2016, we ran the first instance of a massive open online course (MOOC) at Glasgow, providing an introduction to functional programming in Haskell. We received many items of learner feedback that indicated difficulty in learning Haskell. Some people found the tools problematic: *'I have been trying almost all today to get my first tiny Haskell program to run. The error messages on GHCi are very difficult to understand.'* Others struggled conceptually with the language itself: *'[It] is tedious to do absolutely anything in [Haskell] and it made me hate Haskell with a passion.'* Some MOOC participants have tried to learn Haskell several times: *'It's not my first attempt to learn Haskell. I always get stuck with monad part and do-notation.'*

We want to discover how and why novice functional programmers struggle to learn Haskell. What are the common mistakes they make? Once we know the key issues, we can engage with stakeholders such as educators, tool implementers and language designers, to address the problems in various ways.

1. **Improved pedagogy**: More focused textbooks, exercises and online help will provide better support to aid learners to avoid these mistakes. This is why Allen and Moronuki [1] wrote a new textbook based on feedback from years of tutoring Haskell novices. They claim that 'the existing Haskell learning materials were inadequate to the needs of beginners. This book developed out of our conversations and our commitment to sharing the language.'

S. Thompson (Ed.): TFPIE 2017.
EPTCS 270, 2018, pp. 37–51, doi:10.4204/EPTCS.270.3

2. **Error-Aware toolchain**: The standard Haskell tools often feature impenetrable error messages [9]. If we are aware of particular difficulties, then we can provide dedicated support, or customized error messages, to handle these problems. The Helium system [13] is motivated by such considerations.

3. **Modified language**: It may be sensible to lobby the Haskell language committee to remove incidental complexity that causes problems in the first place. For instance, Stefik [25] describes how the Quorum programming language was designed and refined based on empirical user studies. Haskell Prime process activities are similarly user-driven.

But how do we identify the common mistakes that novices encounter? Generally, we rely on first-hand anecdotal evidence. As educators, we think back to how we, as individuals, learned functional programming. However *a sample size of one* leads to the fallacy of *hasty generalization*. If we teach a Haskell course, then we might consider our cohorts of students. The class sizes are relatively small—perhaps tens or at most hundreds of students. Another problem is lack of class diversity. The student population is localized, has a consistent level of of education, and undergoes a common learning experience.

We now live in the era of massive open online courses (MOOCs). MOOC platforms have extensive support for data capture and learner analytics. A MOOC can reach a massive and diverse class, scaling to thousands of students from a range of backgrounds. Table 1 shows the numbers of learners who signed up for various functional programming MOOCs. Public sources are given where available, and the other statistics were obtained from personal emails to the lead educators. The data in Table 1 is presented in the same style as Jordan [15] who surveys general MOOC participation trends.

We designed the learning materials in our MOOC so as to capture learners' interactions via an online system, particularly their interactive programming exercises. Our aim is to analyze the program fragments, to discover what learners find difficult, and whether we might be able to address any of their issues using the strategies outlined above.

The contributions of this paper are as follows:

- We provide an experience report about running a functional programming MOOC, from the perspective of educational practitioners. To the best of our knowledge, this is the first report of this kind.
- We present empirical analysis of learner behavior, showing how the typical MOOC dropoff rate is observed in our interactive coding exercises.
- We advocate a feedback-driven approach to functional programming education, facilitated by analysis of large and diverse learner communities.

The paper structure is as follows: Section 2 describes the interactive coding environment we deployed for novice learners; Section 3 provides a quantitative analysis of learner behavior, based on activity logs; Section 4 outlines threats to validity; Section 5 reviews related work; finally, Section 6 concludes and discusses future work.

2 Evaluation Platform

Our students use a browser-based read-eval-print-loop (REPL) system for the first three weeks of our six week course. This system is based on the *tryhaskell* framework [7] which has an interactive Javascript front-end to parse expressions and provide user feedback, coupled with a server-based back-end that uses

Course title	*Platform*	*Institute*	*Lead Educators*	*Run (year)*	*Wks*	*Signups*	*Compltns*
Functional Programming Principles in Scala	Coursera	EPFL	Odersky, Miller	1 (2012)	7	50K [20]	9.6K
Introduction to Functional Programming	edX	Delft	Meijer	1 (2014)	8	38K [16]	2.0K
Introduction to Functional Programming in OCaml	FUN	Paris Diderot	Di Cosmo, Regis-Gianas, Treinen	1 (2015)	6	3.7K	300
Functional Programming in Haskell	FutureLearn	Glasgow	Vanderbauwhede, Singer	1 (2016)	6	6.4K	900
Functional Programming in Erlang	FutureLearn	Kent	Thompson	1 (2017)	3	5.6K	400

Table 1: Summary of introductory functional programming MOOCs, including statistics where available or confirmed personally (note that completion metrics are not directly comparable across MOOC providers)

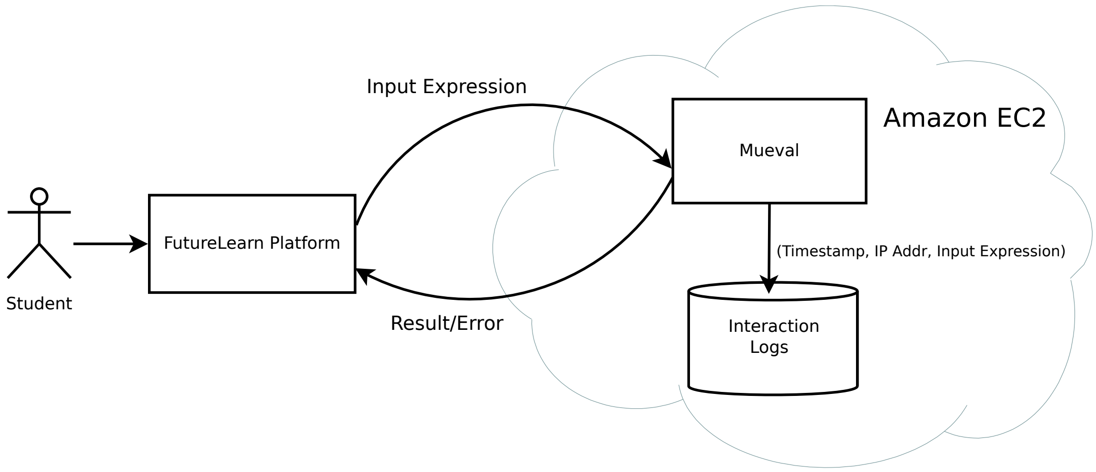

Figure 1: Architectural diagram of our Haskell code evaluation infrastructure, which was hosted on Amazon Elastic Compute Cloud (EC2)

the *mueval* tool [4] to evaluate simple Haskell expressions in a sandboxed environment. Figure 1 shows the architecture of our system. We hosted three load-balanced mueval servers on Amazon EC2 instances for the duration of our course.

At the client-side, a learner follows a series of instructions in an interactive prompt-based Javascript terminal, entering one-line Haskell expressions. See Figure 2 for an example screenshot. These expressions are sent to the server, where they are executed by the mueval interpreter in a stateless way. The result is returned to the client and displayed in the REPL. The mueval interpreter logs each one-line expression it attempts to evaluate, along with a time stamp and an originating IP address.

We make several minor modifications to the original tryhaskell system, which we have forked on github [27].

1. We use Javascript to wrap `let` expressions in a context so we can emulate persistent *name bindings*. This compound `let` expression is automatically prefixed to each one-liner that the user enters, before the code is sent to mueval.
2. Whereas the original tryhaskell system has a fixed sequence of interactions, we allow a *flexible scripted set of interactions* that highlight the particular topics we are covering at each stage of the MOOC.
3. We run multiple evaluation servers concurrently, behind a cloud-based *load-balancer*. A client can be served by any evaluation server, since all interactions are stateless.

3 Learner Data Analysis

In this section, we analyze the data from our MOOC learner population. First we summarize the log files gathered from the three load-balanced servers. The data was collected from 21 September to 3 November 2016, which included the majority of the course. The log files record 161K lines of interactions, comprising around 17MB of data. We logged 3.3K unique IP addresses, which broadly corresponds to the course's 3.1K 'active learners' reported by the FutureLearn platform, i.e. those who completed

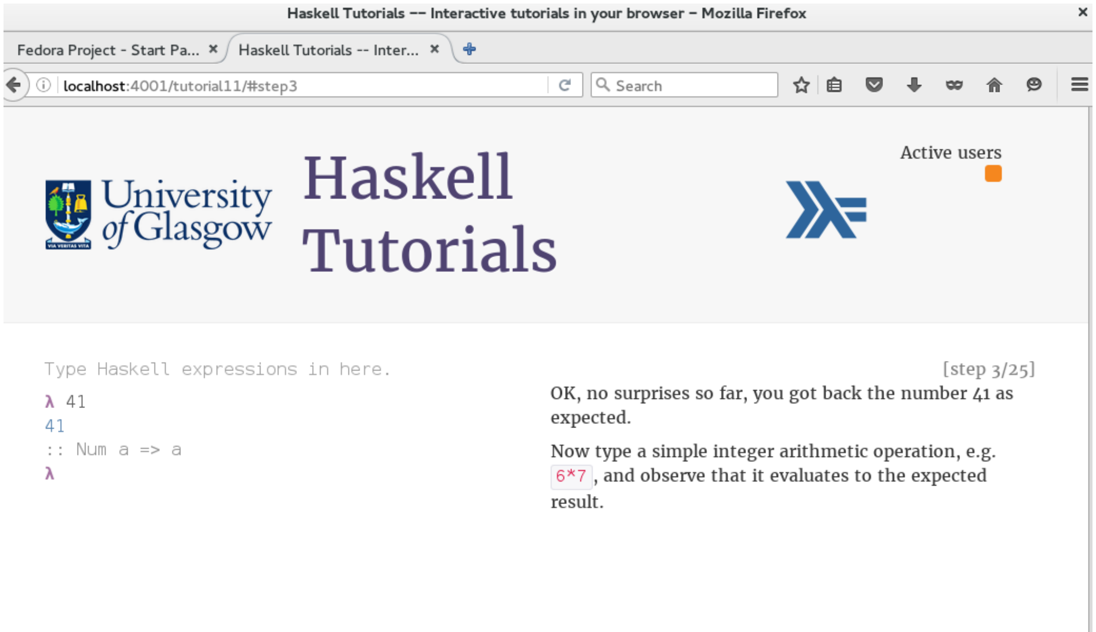

Figure 2: Screenshot of our interactive Haskell tutorial front-end, based on the tryhaskell system

at least one step of the course. Figure 3 shows aggregated counts of these geo-located IP addresses, superimposed on a world map.

Our log files are available on github [24]. We have anonymised the IP addresses by replacing each distinct IP address with an integer value. Otherwise, the log data has not been modified. We encourage other researchers to mine and analyse our logs.

The following sections drill down into more detailed analysis of this data.

3.1 Interactive Sessions

Learners use our interactive tutorial platform in a session-based manner. Each tutorial exercise is designed to take between 5 and 15 minutes, if the user reads the prompts carefully and constructs proper Haskell expressions for evaluation. There are seven exercises in the first three weeks of the course.

From our log files, we attempt to reconstruct these sessions. Hage and van Keeken [12] refer to such sessions as *coherent loggings*. We group together log entries that originate from the same IP address, with an interval of less than 10 minutes between successive interactions. The tutorial system is set up as a read/eval/print loop. As a dual to this, we model the user as a write/submit/think loop. We set a 10 minute limit on iterations round this loop.

We acknowledge that IP address might not be an entirely accurate identifier for a learner, but it is the best proxy we can extract from our log files. Learners were not required to authenticate when they accessed the interactive tutorials.

In total, we identified 5.7K sessions from our logs. The mean number of sessions per IP address is 1.72. This data shows us that many learners did not complete the full set of seven tutorials. Figure 4 shows the number of sessions per IP address, split into 30 equal-sized bins.

The mean time spent in a single session is 490s, or around 8 minutes. This corresponds to our design

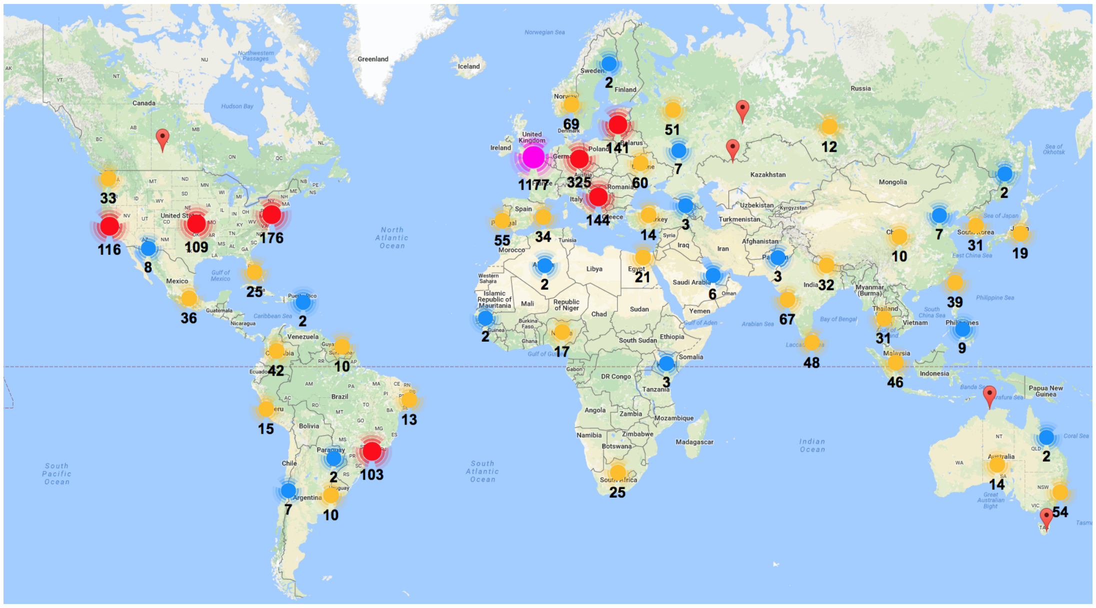

Figure 3: Map of geo-located IP addresses that accessed our servers during the course. Multiple accesses from the same IP address are only counted once in this map. Some IP addresses failed geo-location.

intentions, and also closely follows the observations of Guo et al. on MOOC video length to maximize engagement [11]. Figure 5 shows the length of time per session, split into 30 equal-sized bins.

The mean number of Haskell expressions entered per session is 17. Figure 6 shows the number of lines per session, split into 30 equal-sized bins.

The steep drop-off apparent in these histograms is characteristic of a fat-tailed distribution seen in general MOOC learner engagement, as outlined by Clow with his theory of the *funnel of participation* [5].

3.2 Adventurous Coders

The interactive coding materials, which we host on our forked variant of tryhaskell, are walk-through tutorial style exercises. Each tutorial consists of a fixed number of discrete steps. Each step includes some explanatory text, then the user is prompted to enter a Haskell expression. Sometimes we provide the correct expression directly, and the user simply copies this. Other times we provide hints, and users have to construct their own expressions. Note that we are flexible, in terms of the user entering different code — if the expression is at all appropriate then the tutorial advances to the next step.

We measure the proportion of user input that is based directly on cutting and pasting Haskell code supplied in the tutorial. The rest of the input is modified by users, who have either edited the code to customize it in some way or written something completely different.

In terms of the 161K lines of user input, around 101K are unmodified lines of code and 60K are modified. We also analyze each session individually, to measure the proportion of lines in each session that are modified (i.e. original to that user). Note that sessions have different lengths, as Section 3.1 explains. Figure 7 presents a histogram of sessions, in terms of the proportion of original lines of code per session. The sessions are split into five bins. We observe that the largest bin contains sessions with

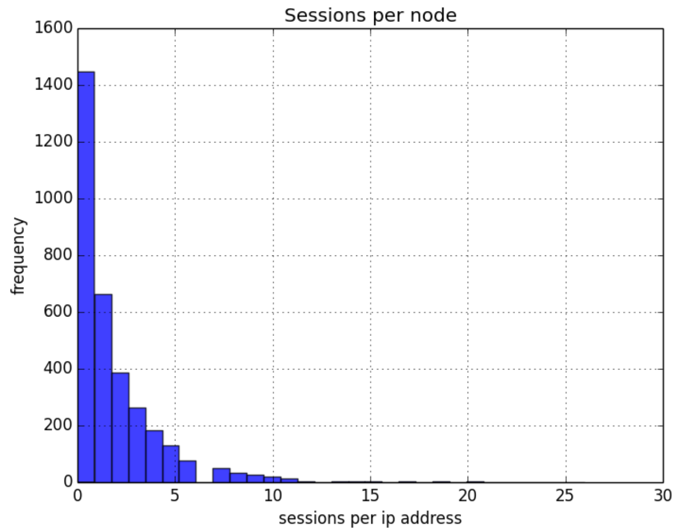

Figure 4: Histogram of sessions split into bins based on number of sessions per IP address

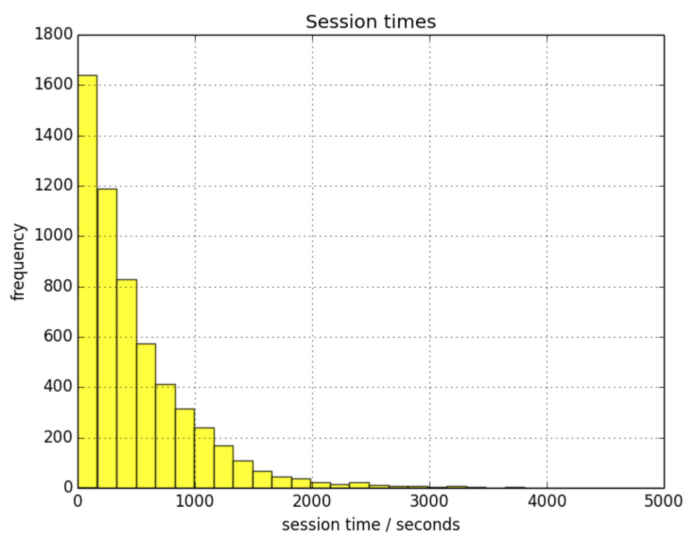

Figure 5: Histogram of sessions split into bins based on the total time of each session

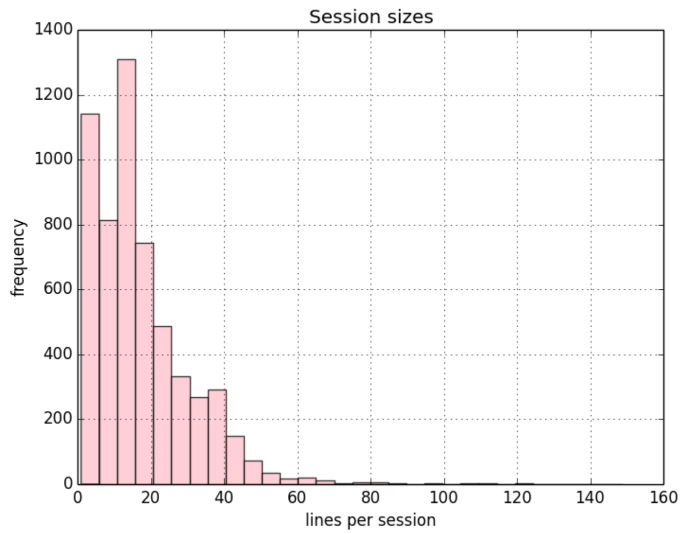

Figure 6: Histogram of sessions split into bins based on the number of lines of code in each session

minimal code modifications.

3.3 Learner Syntax Errors

We extracted the individual Haskell expressions entered by learners, from the log file entries. We ran each of these expressions through a Haskell parser, based on the `haskell-src-meta` package. We discovered that 8.1K of the 161K lines could not be parsed correctly as Haskell expressions.

Of these unparsed lines, 2.1K are parse errors involving the value-naming operator, `=`. In our try-haskell REPL, we accept variable binding operations of the form `var=expr` — which is not a valid expression. A further 390 lines include unexpected `:` characters. These appear to be attempted invocations of gchi commands like `:quit`, which are not supported by the REPL.

Table 2 lists the most common error messages in Haskell expressions, as reported by our parser. We explored these errors by conducting detailed examination of relevant logged interactions. We wanted to determine the high-level root cause of each error. This is a manual task requiring significant domain expertise; we note this does not scale particularly well. However, we identify some apparent high-level problems, such as:

1. *Parenthesis mismatch*: Parenthesis characters are frequently unbalanced. A sample expression that causes this error is: `min((max 3 4) 5))` — Heeren et al. [13] also state that 'illegal nesting of parentheses, braces, and brackets is a very common lexical error'.

2. *Bad scoping*: Problems with `let` and `where` constructs are apparent. The mueval expression parser does not support `where` clauses properly. Some users had confusion with the syntax of `let`, e.g. `let (a,b) (10, 12) in a * 2`.

3. *Misunderstanding do blocks*: Many people tried to bind values to names with `<-` outside a `do` block, or bind names in a `do` block as the final action, e.g. `do putStrLn "nnnnn "; z <- getLine;`

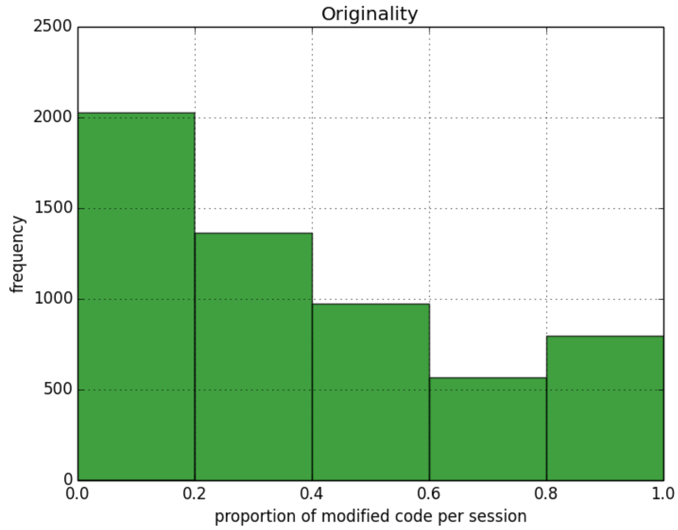

Figure 7: Histogram of sessions split into bins based on proportion of modified lines of code in session

4. *Complex constructs*: The mueval interpreter is particularly restrictive. It does not support `data` or `type` definitions, or definitions of multi-line functions. Several users attempted to enter such code, which did not parse correctly. We encouraged users to transition to GHCi for these constructs. Perhaps some did not read the supporting text, or were expert Haskell users trying to discover the limits of our system.

5. *Incorrect syntax for `enumFromThenTo` syntactic sugar*: People misunderstood the `..` notation, generating incorrect list expressions like: `[0,1,3..10]` or `[0, 2, ..]` or `[1.1, 1.2, .. 2.0]` – this may have been a problem with our tutorial material. Alternatively, learners may be confusing Haskell and Python list syntax.

When learners make mistakes in the programming exercises, we want them to continue to experiment with their Haskell code to fix the problem. The interactive tool provides context-specific feedback and hints for common error cases. We analyze our logs to see how many learners 'give up' when they encounter an error. Of the 5.7K sessions, 3.5% contain a Haskell expression with a parse error as the last line of user input. This suggests a low proportion of learners abandon sessions due to problems with errors.

As course designers, we opted to use tryhaskell because it works 'out of the box' for learners in their familiar browser environments; there is no need to do any confusing or difficult toolchain installation before starting to write code. However we recognize that the tryhaskell environment, with its single line REPL interface and lack of syntax highlighting, is not suitable for learning anything more than the most basic Haskell expressions. We will need to provide more explicit signposting here, or a more fully featured online evaluation environment. In our six week Haskell course, we currently advise learners to transition to GHCi during the third week, when we start to cover more advanced Haskell features that are not supported by the REPL.

Reported error	*Count*
Parse error : }	227
Parse error : ..	223
Parse error : $\mid$	196
Parse error :)	174
Parse error : ,	136
Parse error : <−	135
Parse error : \\	130
Parse error : −>	114
Parse error : in	105
Parse error : ;	98
Parse error : where	91
Parse error :]	85
Parse error : +	64
Parse error : Last statement in a do-block must be an expression	63
Parse error : else	61
Parse error : /	50
Parse error in pattern : length’	45
Parse error : virtual }	41
Parse error : let	36
Parse error : if	34
Parse error : ‘	33
Parse error : {	30
Parse error : ∗	28
Parse error : then	27
Parse error : >	27
Parse error in pattern : l’	26
Parse error in pattern : l	26
Parse error : type	25
Parse error in pattern : f	25
Parse error in pattern :	23
Parse error : data	22
Parse error : =>	21
Parse error : −	20

Table 2: Sorted list of most frequent parse errors for learner input

Expression Type	*Count*
Bool	28798
Num a => a	25198
IO ()	9709
Num a => [a]	9701
[Char]	7589
Int	3910
(Num a, Ord a) => a	3870
(Enum a, Num a) => [a]	3587
IO String	2893
Floating a => a	2361
Num a => a −> a	1840

Table 3: Sorted list of most frequent types observed for learner input expressions

3.4 Expression Typing Analysis

As before, we extracted the individual Haskell expressions entered by learners, from the log file entries. We use the GHC expression evaluation facility to query the *type* of each line of learner input. Of the lines that can be parsed, 28K lines contain expressions with type errors. This is around 18% of all syntactically correct expressions.

We briefly inspected the GHC type error messages to uncover common causes of typing problems. These include many `not in scope` naming problems. Some are simple spelling mistakes, like `lenght`, `putsStrLn` and `xipwith`. Others scoping issues are more complex, where defined names are bound to variables with incorrect types, indicating potential learner misunderstandings of the type system.

We detected further type mismatches via unsatisfiable type constraints, e.g. the error *No instance for (Num Bool) arising from the literal '1'* is caused by a learner attempting an equality test between `True` and `1`. Most of these sorts of errors derive from equality tests and if/then/else expressions. Our interactive lessons encouraged students to explore these facilities, so we should expect such type mismatches.

Table 3 lists the most frequent types observed in learner expressions. We observe that `Bools` are more popular than `Num`eric types. Lists are fairly common. The frequent type constraints are generally induced by `enumFromTo` and if/then/else expressions. The majority of `IO` is due to `putStrLn` and `getLine` which are introduced early in the course. We see that function types are relatively infrequent. Learners can only construct single line function definitions in our REPL environment. Also, the first two weeks of material focus more on using functions (e.g. for list processing) instead of defining them.

4 Threats to Validity

This section considers potential threats to the validity of our findings, in terms of characterizing novice Haskell programmers.

4.1 Lack of Generality

The log files we analyzed were generated by learners from the first run of a single MOOC. While the learners were drawn from varied backgrounds, they were exposed to a single set of Haskell educational resources on this course. In that sense, the findings may not be representative of a wider variety of novices who are taught in a different way, or with different materials.

We acknowledge the need to gather data from repeated runs of the course, and to cross-check this data with other courses. For instance, the Helium project [13] reports similar data, but is at a higher level of abstraction.

4.2 Supported Language Subset

The tryhaskell REPL environment, backed by the mueval utility, handles simple one-line Haskell expression evaluation. In that sense, only a subset of the Haskell language is supported. Hence we are limited to exposing a subset of novice errors.

We argue that the language features supported by mueval are most appropriate for novice Haskell learners — hence the errors are still representative of those that would be experienced in the full Haskell language. We could confirm this by replacing mueval with a more powerful evaluation framework.

4.3 Limited Error Analysis

Our investigation of errors in learner input (Section 3) is a surface analysis. If an expression can be parsed and type-checked, then we do not flag it as erroneous.

We have not fully characterized the type errors, which the Helium developers [13] highlight as being particularly prevalent in novice code. Our REPL does report type errors back to learners when they initially enter the code, but we have not yet classified these errors in a systematic way.

Errors arise from different root causes, e.g. limitations of REPL and learner misunderstandings are common causes. A deeper analysis could identify a wider range of root causes.

4.4 Software Incompatibility

Some learners reported that our interactive tutorials did not work on their machines. This may have been due to web browser incompatibilities or OS issues. The problem seemed to occur mostly on Windows devices (which was a popular OS with users).

Because of this, our logs might be skewed towards users with particular browser/OS configurations, but we do not expect that this will introduce significant bias into the results.

5 Related Work

In addition to the original tryhaskell platform [7] there are similar online REPL systems for Scala (Scastie and Scala fiddle), OCaml (tryOCaml) and other functional languages. While these systems might capture user interactions, to the best of our knowledge there is no publicly available analysis of the data.

Of the four other functional programming MOOCs listed in Table 1, two include integrated interactive coding environments (Scala and OCaml MOOCs). Some survey data from the Scala MOOC is available online [20, 19] although there is no published analysis of learner interactions.

The Helium system [13] logs the behavior of novice Haskell developers. Hage and van Keeken [12] presents some similar metrics and graphs to ours, based on mining data from Helium logs. Their platform performs whole-program execution, so they can work with a richer set of Haskell constructs. They provide a list of common Haskell errors [13]. Some of their lexical errors are identical to ours, such as parenthesis problems. They also have details of common type errors, which we have done in a limited way. They capture and analyze programs created by students at a single institution. Their corpus comprises 30K Haskell programs, in contrast to our 161K one-line Haskell snippets.

Thompson [26] presents a list of common Haskell errors that are generated in the hugs interpreter. Again, there is some overlap with our set of errors.

Bental [3] describes an instance of the interactive Ceilidh framework for assignment-based ML program submission and feedback. She gives a categorization of 134 programming questions submitted by students who could not generate correct ML code to solve particular assignments. 18 of these problems are syntactic — which are most strongly related to the errors we have highlighted in our study.

Marceau et al. [18] present the DrRacket system, and how they devised a set of user-friendly error messages which were the outcome of systematic user trials. They observe that parenthesis matching is a common problem in the first lab exercise, but becomes less apparent in later labs as students gain experience.

DrRacket [8] introduces a LISP-like language to novice programmers via a controlled sequence of gradually more powerful language subsets. It might be possible for us to do something similar with Haskell. We would need to make it clear which language elements are supported in each subset. In effect, the tryhaskell system does define a Haskell subset, since there are many features (e.g. multi-line functions and datatype definitions) that it does not support. It might be sensible to trigger 'wait till later' warning messages if a learner tries to evaluate anything more complex than we expect, inside our REPL. We note that Helium [13] and Elm [6] are deliberately cut-down functional languages that eliminate complex concepts like type classes.

Lämmel et al. [17] advocate a systematic approach to learning Haskell, blended with relevant online content. They present a *chrestomathy*, which is a highly structured set of linked resources, with an accompanying ontology. Their approach introduces distinct programming concepts in a carefully controlled order.

Vihavainen et al. [28] describe an introductory programming MOOC that features scaffolded tasks, which are similar to our Haskell interactive exercises. Their tasks include automated tests, so online learners gain immediate feedback. They also outline the notion of a learning pyramid, where various stakeholders (teachers, tutors, more experienced learners, less experienced learners) form relationships at different levels, in a process they term 'extreme apprenticeship'. Our FutureLearn platform facilitates similar organic interactions between course participants, and we are analyzing this as part of ongoing work.

Murphy et al. [22] note that a small number of UK universities (four) teach Haskell as a first programming language to Computer Science students. This study describes general motivations for selecting particular languages, as well as perceptions of their relative difficulty. However the data does not give particular insights regarding Haskell, since it is such a small proportion of the overall sample.

Large novice programmer datasets are available for other languages, such as Java [2]. This study analyses 18 common errors, of which the majority are syntactic. However it also shows that semantic and type errors are significant, and may take longer to correct.

6 Conclusions

Interactive learning environments are useful for novice developers, but engagement is subject to the standard drop-off that characterizes MOOC participation. We analyzed 161K lines of Haskell code submitted for evaluation by learners, and identified some common syntactic error patterns. Many of these are consistent with previous error classifications reported in the literature.

Based on the errors observed in the learner code, we are targeting specific adjustments to our course materials for the next run of our MOOC. Further analysis may be possible from our log files. For instance, we intend to perform more sophisticated type-based analysis. We have published anonymized versions of our logs on github [24] for the benefit of the research community. We will continue to record and analyze student interactions in future iterations of our Haskell MOOC.

We feel that a richer coding tool (featuring syntax highlighting, parenthesis matching, and multi-line input, *inter alia*) would be more appropriate to support beginner Haskell developers in an online environment. We are currently investigating the IHaskell kernel for the Jupyter Notebook platform [10] as an alternative framework for scaffolded interactive exercises.

Acknowledgments

All learner data was gathered via the FutureLearn platform. Thanks to Wim Vanderbauwhede for co-teaching the Haskell MOOC at Glasgow in 2016. We acknowledge Simon Jouet, who helped with IP geo-location. We are grateful for constructive feedback from Anna Lito Michala, Vicki Dale, Dejice Jacob, and the participants at the TFPIE 2017 conference.

References

[1] Christopher Allen & Julie Moronuki (2017): *Haskell Programming from First Principles*. Gumroad (ebook).

[2] Amjad Altadmri & Neil C.C. Brown (2015): *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data*. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp. 522–527, doi:10.1145/2676723.2677258.

[3] Diana Bental (1995): *Why doesn't my program work? requirements for automated analysis of novices computer programs*. In: *Proc. Workshop on Automated Understanding of (Novice) Programs, World Conference on AI and Education, Washington DC, USA*.

[4] Gwern Branwen (2014): *Mueval*. `https://github.com/gwern/mueval`.

[5] Doug Clow (2013): *MOOCs and the Funnel of Participation*. In: *Proceedings of the Third International Conference on Learning Analytics and Knowledge*, pp. 185–189, doi:10.1145/2460296.2460332.

[6] Evan Czaplicki & Stephen Chong (2013): *Asynchronous Functional Reactive Programming for GUIs*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 411–422, doi:10.1145/2491956.2462161.

[7] Christopher Done (2014): *Try Haskell*. `https://github.com/tryhaskell/tryhaskell`.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler & Matthias Felleisen (2002): *DrScheme: A programming environment for Scheme*. *Journal of Functional Programming* 12(2), pp. 159–182, doi:10.1017/S0956796801004208.

[9] GHC Wiki (2015): *Custom Type Errors*. `https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors`.

[10] Andrew Gibiansky (2017): *A Haskell kernel for IPython*.

[11] Philip J. Guo, Juho Kim & Rob Rubin (2014): *How Video Production Affects Student Engagement: An Empirical Study of MOOC Videos*. In: *Proceedings of the First ACM Conference on Learning @ Scale Conference*, pp. 41–50, doi:10.1145/2556325.2566239.

[12] Jurriaan Hage & Peter van Keeken (2007): *Mining Helium programs with Neon*. In: *Draft Proceedings of the 8th Symposium on Trends in Functional Programming*, pp. 35–50. Available at `http://www.academia.edu/download/30841452/10.1.1.81.4266.pdf`.

[13] Bastiaan Heeren, Daan Leijen & Arjan van IJzendoorn (2003): *Helium, for Learning Haskell*. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pp. 62–71, doi:10.1145/871895.871902.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson et al. (1992): *Report on the programming language Haskell: a non-strict, purely functional language version 1.2*. *ACM SIGPLAN Notices* 27(5), pp. 1–164, doi:10.1145/130697.130699.

[15] Katy Jordan (2014): *Initial trends in enrolment and completion of massive open online courses*. *International Review of Research in Open and Distance Learning* 15(1), pp. 133–160, doi:10.19173/irrodl.v15i1.1651.

[16] Katy Jordan (2015): *MOOC Completion Rates: The Data*. `http://www.katyjordan.com/MOOCproject.html`.

[17] Ralf Lämmel, Thomas Schmorleiz & Andrei Varanovich (2013): *The 101Haskell Chrestomathy: A Whole Bunch of Learnable Lambdas*. In: *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, pp. 25:25–25:36, doi:10.1145/2620678.2620681.

[18] Guillaume Marceau, Kathi Fisler & Shriram Krishnamurthi (2011): *Measuring the Effectiveness of Error Messages Designed for Novice Programmers*. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pp. 499–504, doi:10.1145/1953163.1953308.

[19] Heather Miller (2012): *Visualize statistics from the MOOC "Functional Programming Principles in Scala" using Scala!* `https://github.com/heathermiller/progfun-stats`.

[20] Heather Miller & Martin Odersky (2012): *Functional Programming Principles in Scala: Impressions and Statistics*. `http://docs.scala-lang.org/news/functional-programming-principles-in-scala-impressions-and-statistics.html`.

[21] Randall Munroe (2014): *Haskell*. `https://xkcd.com/1312/`.

[22] Ellen Murphy, Tom Crick & James H. Davenport (2017): *An Analysis of Introductory Programming Courses at UK Universities*. *The Art, Science, and Engineering of Programming* 1(2), p. 18, doi:10.22152/programming-journal.org/2017/1/18.

[23] Simon Peyton Jones (2017): *Escape from the Ivory Tower: the Haskell Journey*. Video–watch from 15:30, `https://www.youtube.com/watch?v=re96UgMk6GQ`.

[24] Jeremy Singer (2018): *Anonymised Logfiles from TryHaskell Servers*. `https://github.com/jeremysinger/haskellmooc_logfiles`.

[25] Andreas Stefik & Susanna Siebert (2013): *An Empirical Investigation into Programming Language Syntax*. *ACM Transactions on Computing Education* 13(4), pp. 19:1–19:40, doi:10.1145/2534973.

[26] Simon Thompson (1999): *Some Common (and not so common!) Hugs Errors*. `https://www.cs.kent.ac.uk/people/staff/sjt/craft2e/errors/allErrors.html`.

[27] Wim Vanderbauwhede & Jeremy Singer (2016): *Haskell Tutorials*. `https://github.com/wimvanderbauwhede/haskelltutorials`.

[28] Arto Vihavainen, Matti Luukkainen & Jaakko Kurhila (2012): *Multi-faceted support for MOOC in programming*. In: *Proceedings of the 13th annual conference on Information Technology Education*, pp. 171–176, doi:10.1145/2380552.2380603.